Table of Contents

Technical Data, Rev. 07.13.06, July 2006
All information subject to change without notice.

US DIGITAL
info@usdigital.com ▪ www.usdigital.com
Local: 360.260.2468 ▪ Sales: 800.736.0194
Support: 360.397.9999 ▪ Fax: 360.260.2469
1400 NE 136th Ave. ▪ Vancouver, Washington ▪ 98684 ▪ USA

1 Product Information

1.1 Features

- Uses full-speed USB mode (12 Mbits/sec)
- Simultaneous acquisition of up to four incremental encoders with up to 24 bit accuracy
- Fully programmable independent counting modes and roll-over values for each encoder
- 32 bit timestamp provided with each encoder sample
- Buffered history of encoder values and timestamps available every 1 ms with low jitter
- Serial Encoder Interface (SEI) bus port for connection to absolute encoders
- Activity LED's show status of each encoder, the SEI bus, and the USB interface
- Eight bits of TTL parallel input can be used as acquisition triggers and storage qualifiers
- Eight open drain parallel outputs, at up to 45V and 150 mA continuous current
- Encoder setup can be saved in non-volatile memory for automatic restore on power-up
- Models available for either differential or single-ended output encoders
- Can trigger on encoder position change, or store only encoder position changes
- Can store position to data logging record within 25 microseconds (max 1kHz rate)

1.2 Description

1.2.1 Encoder Interface

The USB1 tracks up to four encoders with dedicated hardware counters that are individually programmable with respect to count mode, maximum count, quadrature filtering, and indexed reset. The counters are 24 bits wide, and may be programmed to accept quadrature signals in excess of 1 MHz; the default filter setting rejects frequencies above 100 kHz for glitch rejection. The encoder counters are strobed simultaneously so as to provide positions without time skew, and the data is made available every millisecond with nearly indetectable jitter (less than one microsecond).

1.2.2 Parallel Input

The USB1 has eight bits of parallel digital input data, which are sampled and the resulting data stored within a few microseconds of the encoder position data. The parallel input port accepts relay closure signals or TTL input, as well as high voltage inputs up to 30 volts. The signals on the parallel input port may be used as a trigger input for the data logging function.

1.2.3 Parallel Output

The USB1 also incorporates eight bits of open-collector output data, with nominal current capability of 150 mA per output at room temperature. More current may be switched provided that the package maximum is observed (see specifications). It is also capable of withstanding 50 volts of voltage on the output pins.

1.2.4 SEI interface

The USB1 includes an RS-485-like SEI port to allow communication with US Digital's line of absolute encoders.

1.2.5 Timestamp

The USB1 includes a hardware timer that causes an interrupt every 100 usec, and the software increments a 32 bit timestamp value on every interrupt. This is the value returned in the timestamp parameters with encoder position data. The number is an unsigned integer that rolls over every 119.305 hours; note if using Visual Basic it will appear to be a negative number after 59.65 hours, which might be a consideration in the software. If using the timestamp in VB it is recommended that the timestamp end with &H7FFFFFFF& to make it a 31 bit number.

1.2.6 Data Logging Features

1.2.6.1 Overview

The USB1 sends “latest position” data to the host every millisecond USB frame. The driver places this data in a special software register that applications can read, and throws away old data every time new data arrives. However, this data is not guaranteed to be updated every millisecond, and should not be used to record a detailed history of encoder positions; Windows is not a real-time operating system, and occasionally it does not get around to requesting a packet in a given frame because it is busy with some other part of the system. Thus a series of reads might reveal that Windows failed to allow the register to be updated in one or more periods. To avoid this problem, the USB1 provides a separate data logging channel which sends data at a rate requested by the user, and is guaranteed to have every sample present. Up to three records are sent every other millisecond, allowing the driver to catch up with the device when an interruption in data flow occurs. The driver treats this data logging stream in a special way, and buffers it in a dedicated buffer. The data obtained in this manner is not guaranteed to be the very latest position data, but it will have no gaps.

1.2.6.2 Buffering Issues

The USB1 has 96k of on-board buffering, which allows the unit to buffer approximately 4600 samples. This gives 4.6 seconds of data buffering when the sample period is 1 millisecond; longer for longer sample periods. The driver has a buffer for 4096 samples.

1.2.6.3 Sampling Issues

There is a command to set the sampling rate for data logging; this causes decimation of the data within the device, throttling back the stream of data into the driver. The sample rate is a 32 bit unsigned number that represents the number of milliseconds between samples; very long sample periods are possible.

1.2.6.4 Interface

Access to the logging data is through device IO control calls to the driver, or equivalently through function calls to the DLL. The model of interaction is essentially a polled model, and it operates as follows: the driver is called (or DLL) and requests data; if there is data available a flag is set and data is received in the buffer supplied. If there is no new data available then the flag is reset, and no data is placed in the sample buffer supplied. When the program has caught up with the writing process into the buffer there will be no more data until a sample period has elapsed.

1.2.6.5 Triggering

The USB1 may be set up to look for a trigger before storing data via its data logging capability. The trigger may either be a generalized combination of digital input bit conditions, or it may be a change in position of one or more specified encoders. The combination of digital input conditions may include multiple digital input bits, and the conditions may be of different types (high, low, rising edge, falling edge, either edge, always, or ignore). The digital input combinations may be combined with an OR gate or an AND gate.

Alternatively, the USB1 can trigger when one of a specified group of encoders changes position; there is an implicit OR gate in this mode, in which any encoder in the defined group will cause a trigger to occur. The trigger cannot be a mix of encoder changes and digital input conditions, nor can the USB1 trigger on an absolute encoder position. Triggers are always evaluated on one millisecond boundaries, so in general a condition (whether digital or encoder change) must persist for one millisecond to be recognized.

There are commands to start and stop acquisitions; each new start of the acquisition system resets the trigger to begin looking for a new trigger.

1.2.6.6

1.2.6.6 Storage Qualification

Storage qualification is a digital logic analyzer term that describes when samples of data are stored. Like a logic analyzer, the USB1 may be set up to constantly scan its inputs for samples that meet a certain criteria, and then store only those samples. Storage qualification applies only after a trigger has occurred. For instance, the user might set the trigger to start data collection when a machine start signal is detected, and then desire to only store data when a certain sensor is active, indicating some portion of the machine's cycle.

There are three general categories of storage qualification. Two of them, store on generalized digital input conditions and store on encoder position change, are similar to the options available for triggering explained in section 1.2.6.5. Both of these evaluate the conditions on millisecond boundaries, so conditions must persist for at least one millisecond to be recognized.

The third category of storage qualification is to use input IN4 as an interrupt-driven signal that causes low-latency capture of the encoder and digital inputs. No other inputs are used when the USB1 is set up for this mode, and the USB1 captures data within 25 microseconds after a rising edge is seen on IN4. The user must take care to see that successive samples are at least 1 millisecond apart, however. This mode is selected whenever IN4 is set to detect a rising edge, and the other seven input bits are set to Ignore.

1.2.6.7 Caveats

The current software requires 660 microseconds worst case to do all the processing necessary for a single one millisecond USB frame. Commands that are sent to the device while it is collecting data add to that figure, and may cause the device to miss sending data in a frame. Any command dealing with non-volatile storage (read factory, save parameters, read or write of user non-volatile memory) will cause several millisecond frames of data to be dropped. Other simple commands, like encoder reset, or set position are unlikely to cause the miss of a frame, even at maximum sampling rate. At lesser sampling rates (sample periods of 2 millisecond or longer), there will almost never be a problem; even at one millisecond sampling the command would have to arrive at a time when the history channel was trying to catch up after a Windows glitch in order to miss a window. Serial activity on the SEI bus is also expensive in terms of the CPU processor, and may cause lost logging data.

1.2.7 Non-volatile Memory

The USB1 includes non-volatile memory that stores encoder parameters and other data, so that the user can customize the unit to power-up with the encoder resolution and mode that is desired.

There is also a user-accessible area of the non-volatile memory that may be used to store any data. Routines are provided in the DLL for reading and writing this portion of memory. The demonstration program uses this portion of memory to store custom encoder names.

2 Compatibility

The USB1 is compatible with Microsoft Windows 98 / ME, Windows 2000 / XP as a Plug'n Play peripheral. Windows NT and Windows 95 are not supported, neither is the Macintosh OS supported at this time.

3 Installation

3.1 First Time

When the US Digital CD is inserted there is an opportunity to install the USB1 demo program. It is advisable to quit other Windows programs before continuing. Click on "Install Product Software". Choose one of the USB1 software packages from the list of product software. The usual choice is USB1 Demo Software. Labview is available for LabView compatible systems and there is a choice to do development with COM components as well.

If installing a USB1 on the computer for the first time, simply plug in power and connect it to a cable that is plugged into the USB port (Windows 2000 users should be logged in as administrators). Windows will notice that new hardware has been connected, and will open a New Hardware Wizard so that the driver can be installed. Hit Next at the opening screen, and then let Windows search for a driver; just be sure that a floppy or CD containing the driver and the INF file is in the drive, and that the checkboxes next to "search floppy" and “search CD” are checked.

The USB1 manual will be made available as a PDF file from the Start menu, alongside the USB1 Demo program. There is also a zip file provided in the "USB1 Explorer" directory under "Program Files"; this zip file contains the USB1_Explorer source, but be advised it is a fairly large program.

3.2 Upgrading

The USB1_Explorer requires that a USB1 be connected to the Windows 98 / ME or Windows 2000. The installer should be run to re-install the DLL and USB1_Explorer software first.

A copy of the new driver may be found in the root directory of the CD (if you received software in that way). Otherwise after running the installer a new copy of USB1_Explorer a copy will be found in the the directory:
“C:\Program Files\USB1_Explorer\Copies of driver and DLL”. This is the directory to which Upgrade Hardware Wizard must be directed during the Windows 2000 upgrade process described below.

3.2.1 Windows 98 / ME

The simplest way to upgrade the driver in a Windows 98 / ME system is to simply copy the new driver file over the old. Locate the new driver “usdusb1.sys” in one of the two places described in the previous paragraph, and copy it into the “C:\WINDOWS\SYSTEM32\DRIVERS” directory. Windows will warn you that you are copying over an existing file; check to make sure that the file being copied is newer than the existing copy, and press OK to proceed. Remove power from the USB1; wait a few second reconnect power. The new driver will load new firmware into the device, and the new version should be running. If USB1_Explorer is run the firmware revision can be checked by running the Get Factory Info command from the menu.

3.2.2 Windows 2000 / XP

The Windows 2000/XP upgrade process can either be quite involved, or quite simple. To see what situation you are in, open Device Manager (you can find this by right-clicking on My Computer and selecting properties, then Hardware). Plug your USB1 into a USB cable connected to your computer, and apply power to it. Examine the tree-structured list of hardware within Device Manager; your USB1 will either appear under "Encoder Interface" or "USB".

If your device appears under Encoder Interface, the job is relatively simple. Double click on the USB1 device in the tree-structured Device Manager listing, then select the Driver tab. Click on Update Driver. The Update Device Driver Wizard will begin; do not let it search for a driver. Instead, select the second option, "Display a list of known drivers." On the following screen you will see what appears to be a valid driver for the USB1; do not select this because it is the old driver. Instead, insert a disk with the updated driver, and click on Have Disk, followed by Browse. Navigate to the disk (or CD) that you inserted. Select the "usdupdat" or "usdupdat.inf" file, and then click on Open. It will return you to the Install From Disk dialog, where you should click on OK. Now the Wizard should display a list of compatible drivers, and one of them should be titled "Update U.S. Digital USB1 to V1.xx" (where xx is the new revision). Select this driver, and click Next. Click Next and Finish to complete the upgrade as necessary. To complete the upgrade and download new firmware you must cycle power on your USB1.

If your device appears under the USB branch of Device Manager, the upgrade process is quite involved. Contact U.S. Digital Technical support for instructions on how to proceed.

4 USB1_Explorer Demonstration Program

4.1 Installation

The USB1_Explorer requires that a USB1 be connected to the Windows computer system.

The setup program copied files and created directories for USB1_Explorer; there is no need to do any special installation steps. The program may be run from any directory in which there is a copy of the USD_USB DLL file. USB1_Explorer is a compiled Visual Basic program; in rare instances, an update of a DLL in the system may be required in order to run USB1_Explorer. Contact US Digital if this problem is encountered.

4.2 Features of USB1_Explorer

- Point and click simplicity – to change a parameter, click in its text box and type the new value.
- Shows state of all four encoders at once.
- Encoder parameters such as position, maximum count, and mode may be seen and altered by the user.
- Graphical display of input and output parallel data.
- Support for data logging to text file, with pause and resume features.
- Support for sophisticated triggering and storage qualification.
- May be used with multiple USB1 devices.
- Module and encoder names may be assigned to each channel, and are saved in non-volatile memory in the device.

4.3 Description

The USB1 Explorer is a software package that runs under Windows 98/ME and Windows 2000/XP. This program gives the user a graphical, easy to understand window into the USB1 devices connected to a computer. Upon start-up, the USB1 Explorer interrogates the Universal Serial Bus, and compiles a list of which USB1 devices are on the bus. If some of these devices have conflicting module addresses, USB1 Explorer automatically assigns new addresses to these devices. USB1 Explorer then brings up an interface window with controls for each of the incremental encoders and parallel input/output bits on a single USB1. If multiple USB1 devices are present the user may select which device is being displayed from a pull-down menu. The user may simply point and click to change mode information, resolution, or other incremental encoder parameters and output data. Information such as position or parallel input bit status is automatically updated several times per second and displayed graphically. Less frequently used information can be manually requested from the device by the user, and control requests can be sent to the device using simple, pull-down menus. The USB1 Explorer includes a data recording panel which can be used to record position and time data to a text file, and acquisitions may be run unattended (not connected to a computer).

The USB1 Explorer software package includes the program and its Visual Basic source code, the USB1 documentation manual, the USD_USB DLL file that encapsulates and simplifies the interface to the USB1 device, the USB1 device driver, and source code to 2 simpler programs in Visual Basic.

4.4 How to use the USB1_Explorer Program

Make sure that there is at least one USB1 powered and plugged into the computer before starting the program; if more than one USB1 will be used, plug them all in before starting the program. The screen resolution should be at least 800 x 600.

Choose which USB1 to work with by selecting a device by module address in the upper left corner of the screen. When a different device is selected, the program reads the settings from the device and the changes will be reflected in the screen that the USB1_Explorer presented. A different device may be selected at any time provided that an acquisition is not in progress.

To change a channel's position or resolution type directly into white text boxes; changes take effect when the user clicks on another box or control, or when Enter is pressed. Yellow text boxes provide information only, and may not be altered. To make changes non-volatile, exercise the Save Parameters command from the Command menu.
The state of the parallel input data will always be reflected in the colors of the buttons within the frame marked Parallel Input Port; to change output bit states click on the similar buttons below in the frame marked Parallel Output Port.

To acquire a history record of data in a text file, enter a filename, a sample rate, and a number of samples to be collected in the provided boxes, then press the button marked Start (overwrite files); once the acquisition has started that button may be used to pause and resume acquisition. Note that if triggering is being used, the triggering will have to be satisfied on each resumption of data collection.
To use complex triggering and storage qualifier options, press the button marked "Advanced Triggering..." before starting the data logging acquisition. This will bring up a form in which trigger and storage qualifier modes may be specified.

The first choice made should be whether to trigger based on combinations of the digital inputs, or on changes to selected encoder channels. If digital inputs is selected, the eight bits may be set to individually contribute to the trigger specification with the drop menus below. Therefore if it is desired to trigger when the digital input bit 0 goes low, select "Falling Edge" from the IN0 combination box. Any combination of conditions can be selected, and either AND/OR triggering can be used to decide whether they all have to be true simultaneously (AND condition) or whether any individual condition will cause a trigger (OR). To change the AND/OR setting, click on the large button that the eight bit lines lead to.

If choosing to trigger on changes in encoder position, the encoder channels to be evaluated should also be selected; check the appropriate boxes in the lower left most frame at the bottom of the form.

The storage qualification frame may be used to select whether to store when certain digital input conditions occur, or to store only when selected encoder channels change position. The same channels must be used for both triggering and storage qualification if chosen to store only changes in encoder position. It is not possible to trigger or store only when a certain position is reached - the device only provides the general functionality of storing all changes in position.

At the bottom of each frame is a control which can used to set all the bits to a given state at once; this is handy if all bits are to be turned off before turning one bit on. For information on how triggering and storage qualification work, see the description of data logging under section 1.2.6. Section 6.6 also has more detailed information on the triggering system of the USB1; it should be noted that when storage qualification is in effect the period parameter effectively becomes a decimation counter. That is, if a period parameter of 10 msec is shown in the main screen, and the storage qualifier is set to cause storage to occur when bit 0 of the input is rising, then every 10th rising edge will be stored. Most of the time the period parameter should be set the to 1 msec when any storage qualification besides "all bits always" is in effect.

To save encoder parameters, choose Save Parameters from the Command pull down menu; otherwise changes will be lost when the USB1 is disconnected from its power source.

4.5 Caveats

Do not remove USB1 devices from or add USB1 devices to the computer while USB1_Explorer is running; the DLL is certain to return error messages.

The data logging function may generate large files, prudence should be exercised in choosing the sampling rate and number of samples taken. The file will be created or overwritten in the application's home directory (normally C:\Program Files\US Digital\USB1_Explorer).

The sample period text box will convert and round whatever is entered into it to the nearest millisecond, so odd fractions of seconds or hours may not be achievable.

Not all the text boxes have been made bullet-proof; if the user types an alphabetic response into a box expecting a number, it may crash the program with an error.

Note that the trigger and storage qualification functions use data sampled at 1kHz; consequently, a condition must persist for more than one millisecond to be recognized.

Certain parameters are automatically non-volatile; other changes require the Save Parameters command to make the change non-volatile. The following changes are automatically non-volatile: changing the module address, changing the name of the device, and changing the name of a channel. The following parameter changes may be saved via “Save Parameters”: resolution, counting mode, index mode. The output port and the encoder positions are always volatile. It is best not to disturb an on-going acquisition by performing actions that send commands down to the USB1, or the program is at risk of losing/missing data. The USB1 has plenty of bandwidth to handle simple requests, but several controls on the USB1_Explorer interface involve reading or writing to the non-volatile memory, and that ties up the processor for about ten milliseconds.

It is possible to use the USB1 device and the USB1_Explorer program to perform unattended (i.e. not attached to any host) data collection. Here is how to achieve this: set up a triggered acquisition in USB1_Explorer; plan to collect fewer than 4096 samples so as to not overflow the driver’s buffer. Start the acquisition, and verify that it has not triggered yet (look at the title of the data logging frame to see whether it says TRIGGERED or NOT TRIGGERED). Quit the program with the acquisition still not triggered. Unplug the USB cable from the USB1 device; it obtains power from its own power plug, so the acquisition will continue to run and look for a triggering condition. When it finds one, it will collect data according to the storage qualifier that is specified and buffer the resulting data samples, but it will not be able to transfer them to the host because the cable is unplugged. Plug in the USB cable again and the saved data will be transferred to the driver’s buffer; the driver’s buffer may fill, but it will not over-run because a triggered acquisition has finite length. Finally, start up USB1_Explorer again; set up the count of expected samples and the file name in the data logging frame. There is a selection under the Command menu called Read Data From Previously Started Acquisition; choose this command, and the data will be downloaded to the specified file.It may be necessary to choose to Stop And Save File if fewer samples than were expected are acquired.

5 Other Example Programs

5.1 USB Command Tool

The USB Command Tool is a low-level program that interacts directly with the driver. It may be used to send individual commands to the USB1, and examine the responses that come back. Because it uses device IO controls to talk directly to the driver, it runs only under Windows 98. The commands are documented under section 7.1.

5.2 USD_USB.exe

usd_usb.exe is another simple, bare-bones program written in VB, but this one was written especially to exercise the function calls of the DLL. It runs under either Windows 98 or Windows 2000, and source code is provided.

6 Programming with the USD_USB DLL

6.1 Basic Concepts

The USB1 is a device that connects to the Universal Serial Bus, or USB. The USB1 is plugged into the computer's USB port. During power up of the USB1 a functionally specific driver is loaded; this driver is called USDUSB1.sys, and it controls the USB1 by passing commands through the Windows USB class driver. The interface provided by a driver such as USDUSB1.sys is by definition somewhat crude; devices are opened as files, and commands are sent via a mechanism called a device I/O control code (DevIoCtl). To make it easier to write software quickly, a second layer of software is provided that sits between the application program and the driver, in the form of a DLL called USD_USB.dll.

The usd_usb.dll provides a simplified interface to the USB1, usable from either Visual Basic or the C/C++ environment. It is capable of servicing multiple threads of execution, it manages multiple USB1 devices on the bus with ease, and it simplifies the task of associating a given physical device with a software function. It also deals with the fact that the compiler used in the USB1 device firmware stores multibyte numbers in big-endian format, whereas Windows is little-endian; multi-byte results are swapped end for end before being returned to the DLL caller.

The DLL is based on the concept of a Module Address, which is an ID stored within each USB1. The module address is used as a handle to refer to a given physical device when making DLL calls that exchange information with a USB1 in the system. Previous customers of US Digital will recognize the module address as a generalization of the SEI address used in our SEI DLL software. Run one of the supplied demonstration programs to set the module address (which can be any number between 0 and 255) and then mark the module address on the label of the USB1 device. The module address may be set by the software if desired, but more likely it is a function that will be executed only once when the user sets up the system. Once the code has been written to refer to a module address, a USB1 may be replaced if necessary by simply substituting another USB1 that has been programmed with the same module address – no code changes are necessary.

The driver and DLL are inherently multi-threaded safe; more than one application may issue calls to the driver or the DLL, and the driver and DLL will ensure that the commands are properly serialized and sent to the hardware. Multiple copies of the demonstration program may be run they will not crash. However, the demonstration programs are written assuming that they are the only application accessing the USB1, and thus some parameters are not updated on the screen by the timer. For example, if one program writes to the index register, the other program does not know it has happened, and the screen does not reflect the change.

6.2 Getting Started

We have included several demonstration programs that can be used to learn from and be modified for individual applications. The Visual Basic project usd_usb.vbp contains a working program that calls nearly all the DLL functions; this code may be reused as seen fit in applications.

The first thing the software application must do when using the usb_usb.dll is issue a call to USB1Init(). This simple call performs a number of housekeeping tasks: it enumerates all the USB1 devices on the bus, opens a file handle to each, checks for conflicting module addresses, and reassigns any USB1 devices in conflict with each other. It returns a count of USB1 devices found on the bus.

6.3

Next, the software would typically enter a loop that will acquire all the module addresses present in the system. This is done with a call to USB1ReturnModuleAddress(). Like most USD_USB calls this function will return 0 for failure, or 1 for success. If successful, the pointer supplied will have been used to set a module address variable. In VB it might look something like the code below:

```
Dim NumberOfUSB1s as Byte
Dim ModAddress(16) as Byte
Dim I as Integer

NumberOfUSB1s = USB1Init()

If (NumberOfUSB1s > 0) Then
      For I = 0 To NumberOfUSB1s-1
            blnResult = USB1ReturnModuleAddress(I, ModAddress(I))
            If blnResult Then
                  MsgBox "Got good address!"
            Else
                  MsgBox "Got bad Address!"
            End if
      Next I
Else
      MsgBox "No USB1's found!"
End If
```

If there is a single USB1 in the system the loop will only be executed once, and the module address will be given of the single USB1 found on the system. If there are multiple USB1's in the system the software will likely be expecting specific module addresses (that have previously have been set by hand with the demonstration software supplied) so that it can associate given module addresses with specific physical measurements the software is making; if the software does not find the expected module addresses either stop with an error, or perhaps ask the user to enter data (by presenting a question such as Which USB1 Module Address Refers To The Encoders Connected To The Milling Machine?).

Once the software has module addresses the user is free to issue calls to the DLL that return position data, or set up the encoder outputs, or read the digital inputs, etc. Most of these calls are self explanatory; a few, like the call to set up encoder mode, may require more explanation. If using Visual Basic, the source code for the demonstration program includes a Basic module with all the DLL calls listed; this can be found in file USD_USB.bas. If coding in C, the enclosed header and lib files should be used (USD_USB.h and USD_USB.lib, respectively).

6.3 Details of the Encoder Function Calls

Each function call in the DLL is listed below, along with explanatory notes on the parameters, return value, and side effects.

```
int _stdcall USB1Init(void)
```

Enumerates all of the USB1 devices on the bus, resolves module address conflicts, saves module addresses to non-volatile memory if any need to be reassigned, and sets up internal data structures that associate module addresses with device file names. Returns a count of devices found on the bus, zero if none found. Note that devices plugged in after this call is issued will not be known to the DLL, and cannot be accessed through DLL function calls.

BOOL _stdcall USB1ReturnModuleAddress(UCHAR DeviceNumber, PUCHAR pucAddress)
When USB1Init returns a count of N devices found on the bus, the user software must supply an instance number from 0 to N-1 in the DeviceNumber parameter of this call, and the function will use the pucAddress pointer to return the appropriate module address. The function itself returns 1 for success, or 0 for failure.

BOOL _stdcall USB1Close(void)

Closes all file handles associated with USB1 devices enumerated on the bus. Normally, the software will issue this function call just before quitting. Returns 1 for success, 0 if one of the file handle closures failed. Note that this call will fail if the USB device is pulled off the bus after the software issues a USB1Init() but before a USB1Close() is called; there is no harm in this, although Windows 98 will not unload the driver when a device is unplugged with an open file handle. The practical effect of this is that driver updates will not take effect until the machine is rebooted.

BOOL _stdcall USB1GetIncPosition(UCHAR ModAddress, UCHAR Encoder, LONG* ulResult)

Queries the given USB1 with the given encoder channel (0 to 3) for position, which is returned as a 24 bit unsigned integer within a 32 bit long (via the last parameter, which is a pointer). The function returns 1 for success, 0 for failure.

BOOL _stdcall USB1GetAllPosition(UCHAR ModAddress, ULONG* pulPosition, ULONG *pulTimeStamp, UCHAR *pucParallelInput)

This function returns a coherent snapshot of all the encoders and the digital inputs of a given USB1, along with a timestamp. The pulPosition parameter should be a pointer to an array of four long integers to receive the positions of the encoders, the pulTimeStamp parameter is a pointer to an unsigned long timestamp variable, and the pucParallelInput parameter is a pointer to single byte variable to hold the read of the input port. Returns 1 for success, 0 for failure.

BOOL _stdcall USB1GetIncPositionExtended(UCHAR ModAddress, UCHAR Encoder, LONG* lPos, LONG* lMaxCount, LONG* lTimeStamp, PUCHAR pucCounterMode, PUCHAR pucStatus)

This function is similar to the USB1GetAllPosition() call, except that it includes pointers to store information about all the operating parameters of an encoder. These include what the maximum count value is for this channel (the roll-over value), the counter mode byte programmed into the LS7266R1 chip, and the flag status byte returned by the LS7266R1 chip. For information on how to interpret these values, refer to the LS7266R1 data sheet available on www.usdigital.com.

BOOL _stdcall USB1ResetIncPosition(UCHAR ModAddress, UCHAR Encoder)

This function call resets the position of the specified encoder on the specified USB1 module address to zero. Returns 1 for success, 0 for failure.

BOOL _stdcall USB1ResetAllIncPositions(UCHAR ModAddress)

This function resets all the encoders on a given module address to zero; the command is issued simultaneously to all four encoders without latency. Returns 1 for success, 0 for failure

BOOL _stdcall USB1SetIncPosition(UCHAR ModAddress, UCHAR Encoder, ULONG Data)

This function call sets the specified encoder channel (0 to 3) on the specified module address to the specified position. Note that the call will fail if the software attempts to set the position greater than or equal to the currently specified roll-

over value (max count) of the channel. The position supplied is an unsigned positive integer, and attempting to set the position to a negative number will not work. The counter that tracks encoders in hardware reads out in postive integers only. Returns 1 for success, 0 for failure.

BOOL _stdcall USB1SetIncMaxCount(UCHAR ModAddress, UCHAR Encoder, ULONG Data)

This function sets the maximum count on the specified encoder channel (0 to 3) on the specified module address. The number loaded into the counter's roll-over (preset) register is actually one less, so if a maximum count of 50 is specified the counter will display count 48, 49, 0, 1, 2, etc; there will be fifty unique codes out of the encoder channel, but the value of 50 itself is never seen. The counter roll-over register is 24 bits wide; any number between 1 and 2 to the 24th power (0x01000000) may be specified, though of course a max count of 1 is not very useful - zero is all that will be read out of the counter. Typically a max count that matches the encoder resolution would be entered. Returns 1 for success, 0 for failure.

BOOL _stdcall USB1SetIncMode(UCHAR ModAddress, UCHAR Encoder, UCHAR Mode, UCHAR Prescale)

The mode byte is what the firmware sends directly to the encoder channel's LS7266R1 mode register; typical values are 0x2E for x1 counting mode (the counter increments once for every four quadrature changes, or once for every full quadrature cycle), or 0x3E for x4 counting mode (in which the counter increments on every quadrature change, or four times per full quadrature cycle). For more details consult the LS7266R1 data sheet available on www.usdigital.com. The prescale byte controls the digital filtering that the counter chip applies to its quadrature inputs; the table on the next page lists some full-cycle cut-off frequencies for various prescale values. Returns 1 for success, 0 for failure.

Prescale Value	Max. Input Frequency
0	3 Mhz
1	1.5 Mhz
2	1 MHz
3	750 kHz
4	600 kHz
6	428 kHz
9	300 kHz
14 (0x0E)	200 kHz (default)

BOOL _stdcall USB1GetFactoryInfo(UCHAR ModAddress, USHORT *pusModel, USHORT *pusVersion, USHORT *pusConfiguration, ULONG *pulSN, UCHAR *pucMonth, UCHAR *pucDay, USHORT *pusYear)

This function call is passed pointers to return factory parameters specific to the USB1 at the module address specified. USHORT parameters are 16 unsigned short integers. The most useful parameter for customer applications is the pointer to the serial number. Returns 1 for success, 0 for failure.

BOOL _stdcall USB1SetIndexRegister(UCHAR ModAddress, UCHAR ucNewIndex)

The USB1 may be programmed via this call to reset an encoder channel's counter to 0 when the encoder asserts a positive-true index signal in hardware. Only the lower 4 bits of ucNewIndex are used; bit 0 should be set high to make channel zero utilize index reset, bit 1 for channel 1, etc. Sending a 0 byte for ucNewIndex disables all 4 channels from using index reset. Note that non-index encoders typically output random signals on their index pins, which can cause erratic behavior on a USB1 if a channel for index reset is enabled but fails to make sure that the encoder connected to that channel is truly an indexed encoder. Returns 1 for success, 0 for failure.

```
BOOL _stdcall USB1GetIndexRegister(UCHAR ModAddress, PUCHAR pucIndex)
```

The specified USB1 at the supplied module address is read, and the pointer pucIndex is used to update a variable with the current setting of the unit's index enable register. Only the lower four bits are valid; bit zero set high means channel 0 is enable to use index reset, bit 1 for channel 1, etc.

```
BOOL _stdcall USB1SaveParameters(UCHAR ModAddress)
```

This function call saves the operational parameters (but not the counts) of the four encoder channels to non-volatile memory so that the next time the specified USB1 unit is powered up the operational state of the unit will be restored. The parameters saved include: the mode byte and the maximum count for each encoder, the index enable register, and the module address. Note that issuing this command makes the USB1 unavailable for a short period of tens of milliseconds, which will create a gap in the data returned from the device; do not send this call at a time when it is critical that the software be able to read data from the device. Also take care not to exceed the EEPROM's 100,000 cycle wear-out specification. Returns 1 for success, 0 for failure.

6.4 Details of the Input/Output Port Function Calls

```
BOOL _stdcall USB1GetInputPort(UCHAR ModAddress, PUCHAR pucResult)
```

This function call uses the pointer pucResult to return the current data present on the USB1 input port; the port has pull-ups on it, so unconnected bits will be read as 1's. If a timestamp is needed with the data, use the USB1GetAllPosition() function call. Returns 1 for success, 0 for failure.

```
BOOL _stdcall USB1SetOutputPort(UCHAR ModAddress, UCHAR Data)
```

This function call sets the output port on the specified USB1 to the byte value supplied in parameter Data. The output port is "open collector;" setting a bit to 1 turns on the corresponding transistor, which causes it pull down (sink current) on the output pin. Returns 1 for success, 0 for failure.

```
BOOL _stdcall USB1GetOutputPort(UCHAR ModAddress, PUCHAR pucOutput)
```

This call provides a means to read back the state of the parallel output port. The parameter pucOutput receives the result; bits set are bits energized to pull down to ground.

```
BOOL _stdcall USB1SetOutputBits(UCHAR ModAddress, UCHAR Data)
```

This call is similar to USB1SetOutputPort(), except that it is not necessary to know the current state of bits that the user does not want to affect. The user would use this call when he wants to turn on 1 specific bit, and leave all the other bits alone; the data supplied is OR'ed with current state of the output port. The resulting high bits turn on open collector outputs.

```
BOOL _stdcall USB1ClearOutputBits(UCHAR ModAddress, UCHAR Data)
```

This call is similar to USB1SetOutputPort(), except that it is not necessary to know the current state of bits that the user does not want to affect. The user would use this call when he wants to turn off 1 specific bit, and leave all the other bits alone; the data that is supplied is inverted and AND'ed with current state of the output port.

6.5

6.5 Details of Miscellaneous Housekeeping Function Calls

BOOL _stdcall USB1SetModuleAddress(UCHAR ucOldModAddress, UCHAR ucNewModAddress)

This function call sets the module address of the specified USB1 to the new address specified in ucNewModAddress. The change is volatile, and must be followed by a call to USB1SaveParameters() to make it permanent. The DLL will notice that the same physical device now has a new address, and subsequent DLL calls may use the new address. If the software attempts to set the address to an address already in use by one of the USB1's on the USB port the call will fail. Returns 1 for success, 0 for failure.

BOOL _stdcall USB1ReadUserEEPROM(UCHAR ModAddress, ULONG ulLocation, ULONG ulLength, PUCHAR pucData)

The USB1 provides a 64-byte area of the non-volatile memory for applications to write and read from as they choose. The ulLocation and ulLength parameters define the starting position (0-63) and length (1-64) for the read, which is put in the array of bytes pointed to by pucData. The read operation does not wrap around; the DLL will reject reads that try and read location 64 and greater. No structure is imposed on the data that the user reads or writes; the user is free to define the use of this area as desired. The USB1_Explorer program uses this area as storage for encoder and module names; the module name is 19 bytes, encoder names are 11 bytes, and the remaining byte at location 63 is a checksum of the user area.

BOOL _stdcall USB1WriteUserEEPROM(UCHAR ModAddress, ULONG ulLocation, ULONG ulLength, PUCHAR pucData)

Similar to the call above, only data is written to the user area. Note that the EEPROM is only guaranteed to operate for 100,000 write cycles.

BOOL _stdcall USB1GetLastErrorVB(LPSTR pszDestination, size_t lngMaxCount);;BOOL _stdcall USB1GetLastErrorAnsi(LPSTR pszResultString, ULONG ulLength)

The DLL includes an error logging facility, and these function calls return information on the last error seen by the DLL. The call also clears the message concerning the last error seen. The Ansi version of the error retrieval function is intended for use with C programs that use single-byte character sets (non-Unicode). The VB version is for use with Visual Basic, and requires that the function be declared with the first parameter as ByVal which is counter-intuitive, but works. See Microsoft Knowledge Base article Q187912. The resulting strings returned attempt to be descriptive where possible; if an error is encountered in something that does not require a parameter, or the parameter seems to be correct, contact US Digital support staff to report a bug. The largest error string that may ever be encountered is 512 characters long. The function returns 1 for success, 0 for failure, and should not fail.

BOOL _stdcall USB1SetTimeStamp(UCHAR ModAddress, ULONG ulNewTS)

This call may be used to set the internal timestamp counter of the USB1 to any arbitrary 32-bit value. Typically a user would call this function with a second parameter of zero to reset the timestamp counter to zero.

6.6 Data Logging DLL Calls

This version of the DLL now has basic support for the buffered history function of the USB1. In this version of the DLL the user application is able to read out single encoder and all encoder position information from the buffered history subsystem of the driver. The data is sampled as frequently as every millisecond by the device with minimal jitter (the timestamps never change in the last digit), and buffered first on the device and then again within the driver.

The DLL currently supports polled access to these functions; both of the data readout DLL calls include a pointer to a boolean variable, pbDataAvailable; when the DLL sets this variable true it means that there was data available, and variable pointed to by the position/timestamp/input byte pointers has been updated. If the boolean variable has been set false it means that the application has completely caught up with the writing function, and the buffer is currently empty.

The DLL also offers access to the triggering and storage qualification capabilities of the USB1. The user can supply a data structure that defines when an acquisition triggers, and when subsequent data is stored. When trigger/storage qualification information is supplied the USB1 enters a triggered acquisition mode; this means that from that point on whenever an acquisition is started the trigger will first have to be satisfied before data will be examined for storage, and even after the unit has triggered each and every sample that is a candidate for storage it will be compared against the storage qualification criteria. Only samples meeting the criteria specified will be stored. This two-stage acquisition control gives the user powerful control over the USB1 to collect data. For instance, the user might wish to trigger on the start of a certain machine cycle, and only store data when a certain sensor indicates a portion of the mechanism is in a certain range.

The USB1 powers up with a non-triggered and non-qualified acquisition in progress; thus the buffers can simply be flushed to clear out accumulated data, and begin reading data if desired. The default data stream is at full 1kHz rate. The two data read-out DLL calls are listed below:

```
BOOL _stdcall USB1GetBufferedPosition(UCHAR ModAddress, UCHAR Encoder, LONG* ulResult,
BOOL *pbDataAvailable)
```

Similar to USB1GetIncPosition(), except returns history data from the driver via ulResult pointer. The last parameter, a pointer to a Boolean variable, will be set true if there was data available; if false, it means that the application has completely caught up with the driver buffer, and there is no more data yet. Note that using this call consumes all four encoder positions in a given sample. This call should not be used to read out all four encoders, because the driver read pointer increments on each access of the buffer.

```
BOOL _stdcall USB1GetAllBuffered(UCHAR ModAddress, ULONG* ulPosition, ULONG
*ulTimeStamp, UCHAR *ucParallelInput, BOOL *pbDataAvailable)
```

Similar to USB1GetAllPosition(), except returns history data from the buffered subsystem in the driver via the parameter pointers for position, timestamp and input byte. The last parameter, a pointer to a Boolean variable, will be set true if there was data available; if false, it means that the application has completely caught up with the driver buffer, and there is no more data yet. The following two DLL calls provide access to set and read trigger/storage qualification to/from the USB1:

```
BOOL _stdcall USB1SetEvents(UCHAR ModAddress, PUCHAR pucTrigger, UCHAR ucTrigAnd,
PUCHAR pucQualifier, UCHAR ucQualAnd, ULONG ulNumberOfSamples)
```

The pucTrigger parameter in this call is a pointer to an eight byte array of triggering codes. Similarly, the pucQualifier parameter is a pointer to an eight byte array of qualifier codes. Each byte in the array refers to a bit on the parallel input port, beginning with bit 0. The software should set up each byte of these arrays with a code from the table on the next page:

Trigger or qualify never (ignore)	0
Trigger or qualify on rising edge	1
Trigger or qualify on falling edge	2
Trigger or qualify on either edge	3
Trigger or qualify on high condition	4
Trigger or qualify on low condition	5
Trigger or qualify unconditionally (always)	6

The parameter ucTrigAnd controls whether the preceding array of conditions is to be AND'ed or OR'ed together by the trigger system; a non-zero value specifies an AND condition, and 0 specifies OR. The ucQualAnd parameter is similar, only it controls the storage qualification condition codes. The ulNumberOfSamples parameter specifies the number of samples; supplying a 0 in this position indicates that the acquisition is intended to continue indefinitely.

The data transmitted in the eight byte arrays also determines the triggering and storage qualification methods that the USB1 will use when collecting data. For triggering or storing on position changes, all eight bytes of the trigger or storage qualification array should be set to 'never'. This causes the firmware to look for changes in encoder position rather than digital inputs; the active channels that will be scanned should be specified using the USB1SetPositionQual() function call.

There is also a special combination of bytes used when the user wishes to employ the low-latency interrupt-driven data acquisition feature of the USB1. This feature only applies to storage qualification, not triggering. To choose this feature, all the channels except IN4 should be set to 'ignore', and digital input IN4 should be set to 'rising edge.' See section 1.2.6.6 for more background on this feature.

If neither of these two special byte array conditions is created, the USB1 uses the bytes of the array to define a generalized trigger or storage qualification based on the digital input bits.

```
BOOL _stdcall USB1GetEvents(UCHAR ModAddress, PUCHAR pucTrigger, PUCHAR pucTrigAnd,
PUCHAR pucQualifier, PUCHAR pucQualAnd, PULONG pulNumberOfSamples)
```

The data parameters of this call mirror those of the call above that sets the trigger/storage qualification events; the main difference is that all the parameters here are pointers, so the DLL may return values for the trigger and storage qualification events.

```
BOOL _stdcall USB1SetHistoryBufferRate(UCHAR ModAddress, ULONG ulRate)
```

The ulRate parameter defines how the 1 millisecond raw acquisition data will be decimated; if the user supplies a value like 5, then every fifth sample will be buffered and transmitted to the host, resulting in an effective sampling rate of 1000/5, or 200 Hz. If using storage qualification, every fifth qualified sample will be buffered and sent. This command may even be invoked during an acquisition in progress; it will take effect on the next reload of the decimation counter.

```
BOOL _stdcall USB1StartAcquisition(UCHAR ModAddress)
```

This call resets the trigger and begins an acquisition on the specified USB1. Data will not be transmitted to the host driver until the trigger condition is met, and the rate at which data arrives will be determined by the sample rate set by USB1SetHistoryBufferRate().

BOOL _stdcall USB1StopAcquisition(UCHAR ModAddress)

This call terminates an acquisition on the specified USB1; no further samples will be buffered, though remaining samples in the USB1 buffer will be sent to the host.

BOOL _stdcall USB1FlushHistoryBuffer(UCHAR ModAddress)

This call flushes the USB1's history buffer, effectively throwing away whatever samples have accumulated. Both the driver buffer and the device buffer are flushed.

BOOL _stdcall USB1GetHistoryBufferStatus(UCHAR ModAddress, PUCHAR pucStatus)

In rare instances, it might be possible for the USB1 hardware buffer to be overrun. This command returns SUCCESS (0x01) if everything is OK, or FAILURE (0x00) if the host has not kept up with the stream of data from the device. Reading the status via this call clears the error condition.

BOOL _stdcall USB1GetAcquisitionStatus(UCHAR ModAddress, PULONG pulSamplesToCollect, PULONG pulSamplesRemaining, PUCHAR pucStatus)

The pulSamplesToCollect and pulSamplesRemaining parameters are pointers to unsigned longs that describe the number of samples set for the current acquisition, as well as the number of samples remaining. The pucStatus parameter is a pointer to a unsigned byte (UCHAR) that has the following encodings: bit zero is set if an acquisition has been started and is in progress, bit 1 is set if the acquisition has been triggered, and bit 2 is set if the acquisition is of indefinite length (continuous).

BOOL _stdcall USB1GetPositionQual(UCHAR ModAddress, PUCHAR pucQualByte)

The pucQualByte parameter is a pointer to an unsigned char that defines which encoder channels contribute to triggering and storage qualification when the digital inputs are all set to Ignore/Never. Bit 0 enables channel 0's contribution, bit 1 for channel 1, etc. The default power-up value of this parameter is zero, indicating that no encoder channels are enabled for triggering or storage qualification. Note that the same definition is used for both triggering and storage qualification; it is not possible to trigger exclusively on one channel, and store exclusively on another.

BOOL _stdcall USB1SetPositionQual(UCHAR ModAddress, UCHAR ucNewQualByte)

This call is used to set the byte that controls which encoder channels are used in either triggering or storage qualification. To enable an encoder channel to participate in either triggering or storage qualification, set the corresponding bit in the byte that you supply with this call (bit 0 for channel 0, bit 1 for channel 1, etc). Note that there are two steps for successfully setting up an encoder channel as a trigger or storage qualification; one is to set which encoder channels participate, using this call. The other requirement is that the definition of the digital input bits for triggering or qualification must be set to 'Ignore'; having all digital input trigger channels set to ignore causes the processor to use the enabled encoder channels for triggering. Similarly, setting all digital input bits qualification status to Ignore causes the processor to use the enabled encoder channels for storage qualification.

6.7

6.7 SEI DLL Calls

US Digital is in the process of adding support for SEI operations. The current DLL includes preliminary calls for these functions, but more functionality will be released soon. The currently supported calls include:

```
BOOL _stdcall USB1CommandResponseSEI(UCHAR ModAddress, PUCHAR pucCommand, ULONG
ulCommandLength,PUCHAR pucResponse,PULONG pulResponseLength)
```

This is the basic call for exchanging information with the SEI bus. The pucCommand is a pointer to a byte string (array) that contains the sequence of bytes that should be sent out on the SEI port of the USB1 at module address ModAddress. The ulCommandLength tells the DLL how many bytes are in the string. The pucResponse parameter is a pointer to a byte array where the driver should place the response back from the SEI device; this array should have room for at least 64 bytes, the maximum number of bytes that can be returned. The pulResponseLength parameter is a pointer to an unsigned long; when this routine is called the pointer should point to a variable initialized to the size of the buffer (normally 64). Upon completion of the call, the DLL will update this variable through the pointer to show the number of bytes returned.

Some caveats: recall that SEI devices return multibyte data in big-endian (Motorola) format, so the user will have to swizzle bytes to bring numbers into Intel format. The driver assumes that every SEI command will generate some kind of response; if it sends out a command, and does not get a response, this DLL call will return boolean failure. In fact, the command may have succeeded, but the fact that no response came back causes a reported failure. Do not try to send more than one SEI command in each call; the SEI bus is half-duplex, and the response from the first command will collide with the sending of the second command.

Example: suppose a single A2 is connected to the SEI port of the USB1, which resides at module address 16. In order to obtain position information from the A2. The A2 position command is a single byte (something like 0x12), where the second nibble of the byte (2 in this case) is the SEI address. Two bytes are normally received back as a response. The following code fragment illustrates how to read position from an A2 connected to the SEI port of a USB1:

```
ULONG lngGetA2Position(UCHAR ucModAddress, UCHAR ucSEIAddress)
{
                UCHAR ucCommandBuff[64];
                UCHAR ucResponseBuff[64];
                ULONG ulResponseLength = 64;

                ucCommand[0] = (0x10 | (ucSEIAddress & 0x0F));
                if USB1CommandResponseSEI(ucModAddress,
                                ucCommandBuff,
                                1,
                                ucResponseBuff,
                                &ulResponseLength) {
                if (ulResponseLength == 2) {
                     return(ulResponseBuff[1]+(256*ulResponseBuff[0]));
                }  // end if response length is 2
                }  // end if command succeeded

                return(0xffffffff);   // else return invalid value
}
```

This method will allow the user to send almost all the useful commands to any SEI device; the only unsupported commands are those that withhold or extend the BUSY status.

```
BOOL _stdcall USB1GetSEIStatus(UCHAR ModAddress, UCHAR *Result)
```

This call accepts a module address and a pointer to a status byte variable. The status byte will update as follows: bit zero is set if a framing error has been encountered, bit one is set if a buffer over-run condition occurred within the USB1, and bit two is set if the BUSY status lines of the SEI interface are currently asserted. Reading status with this function call resets the internal status of the USB1 for framing error and buffer over-run (if the status is read twice in quick succession the second call will return no error on these two bits). The function returns TRUE if successful, and FALSE if an error occurs.

```
BOOL _stdcall USB1ResetSEI(UCHAR ModAddress)
```

This function call will cause the SEI bus on the specified USB1 to be set to the break condition for one second, which will reset all SEI devices on the bus.

```
BOOL _stdcall USB1SetSEIBaudRate(UCHAR ModAddress, ULONG ulBaudRate)
```

This function call sets the SEI port of the specified USB1 at ModAddress to a new baud rate. The allowed values for the ulBaudRate parameter are 2400, 4800, 9600, 19200, 38400, and 57600; no other baud rates are supported, even though most SEI devices are capable of using 115 kbaud. Note that the user must be careful in how he sequences the change of baud rate; the best way is to issue a single SEI command (via USB1CommandResponseSEI) setting all the devices to a certain baud rate, and then issue a call to USB1SetSEIBaudRate to change the SEI host baud rate.

6.8 Caveats

The current implementation of USB_DLL has a limitation when it comes to multiple threads/applications using the DLL: it only queries the operating system for the presence of USB1's when the first process calls USB1Init(). Thus, whatever USB1's are plugged into the computer at the time of that call are the ones that get recognized; if another USB1 is added to the system with an application open that uses the DLL, it won't get recognized, even if that second application calls USB1Init() after the plug-in occurs. A later version of the DLL may have a more graceful way of handling this, but for now the limitation exists.

Users of Visual Basic should be aware of a potential trap that exists with respect to return status codes. Visual Basic uses 0 for False, and –1 for True, whereas the DLL returns either 0 or 1. Be wary of using expressions like

```
If Not USB1GetIncPosition(...) Then
```

Because VB will simply complement all the bits of a Success result (0x01) returned by the DLL to form (0xFE), which will then test True again because it is not zero. Do not compare function call return values against True or False, either, because they may fail in unexpected ways. DLL function calls can be tested for success using

```
If USB1GetIncPosition(...) Then
```

VB checks to see that the resulting return from the function call is non-zero. If action needs to be taken based on a function call that failed, use a construct with an empty true clause like:

```
If USB1GetIncPosition(...) Then
      'do nothing
Else
      'do something constructive here
End If
```

7 Special Topics

7.1 Hardware Specifications

Parameter	Min.	Typ.	Max.	Units	Notes
Supply voltage	8.0	-	16	Volts	Compatible with **PS-9V.**
Supply input current	-	50	-	mA	@ 12VDC w/ no current drawn by encoders or external I/O.
Regulated output voltage for external encoders and I/O	4.8	-	5.2	Volts	0 to 350mA output current, 25°C, for 8 to 16 VDC power input.
Current available for	-	350	-	mA	Using **PS-9V**.
powering encoders and I/O	-	750	-	mA	Using 12V, 1.2 amp min. DC power source.
Open collector maximum voltage, Out0 - Out7 to ground	-0.7	-	50	Volts	Clamped by driver IC at 50V and clamped to ground by internal diode.
Open collector sink current (sum of)	-	-	1.2	amp	Continuous @ 25°C.
Out0 - Out7 (package maximum)	-	-	720	mA	Continous @ 70°C.
Open collector sink current, any single output Out0 - Out7	-	-	450	mA	Continuous, over temperature range.
Open collector saturation voltage	-	0.5	-	Volts	Any single output, 100 mA sink current, 25°C
Open collector reverse current	-	-	500	mA	Continous, output is clamped to ground by internal diode when external conditions try to force output negative (flyback).
TTL input port high level input voltage (Vih)	2.0	-	30	Volts	In0 - In7 are diode clamped to +5 volt internal through a 10k resistor.
TTL input port low level input voltage (Vil)	-0.5	-	0.8	Volts	In0-In7 are diode clamped to ground through a 10k resistor.
TTL input port high level input current	-	10	-	uA	
TTL input port low level input current	-	-500	-	uA	In0-In7 have internal 10k pullups to +5 Volt internal supply.
Encoder input, high level input voltage Vih	2.0	-	-	Volts	Compatible with 5V TTL optical encoders. See note 2.
Encoder input, low level input voltage Vil	-	-	0.8	Volts	Compatible with 5V TTL optical encoders. See note 2.
Encoder input, differential model, common mode voltage	-7	-	7	Volts	See note 3.
Encoder input, differential model, high level input current	-	0	-	mA	When driven by TI 26C31; the USB1 uses a TI 26C32 differential receiver, which is compatible with US Digital line drivers and 5V differential TTL output encoders.
Encoder input, differential model, low level input current	-	2.7	-	mA	When driven by TI 26C31.
Encoder input, single-ended model, high level input current	-	-0.2	-	mA	When driven by 5V TTL optical encoder; the USB1 uses a TI 26C32 differential receiver, and is compatible with 5V TTL output encoders.
Encoder input, single-ended model, low level input current	-	2.7	-	mA	When driven by 5V TTL optical encoder.
Quadrature cycle input frequency	0	-	1.5	MHz	Maximum spec. requires software command to set; default as shipped is 100KHz.
SEI port high level output voltage	-	4.75	-	Volts	The SEI port is compatible with US Digital absolute encoders; see **A2** data sheet.
SEI port low level output voltage	-	0.1	-	Volts	
SEI port baud rates	2400	-	57.6k	baud	Default is 9600 baud.
SEI port: common-mode input voltage	-7.0	-	12	Volts	

Notes (for chart):

1. Parallel output port is actually a MOSFET driver connected in an open drain configuration; specifications have been translated and given in the more familiar open-collector nomenclature. Note that setting a port bit to 1 turns on a transistor to sink current to ground.
2. Single-ended input model is terminated with 2.35k pullup to +5V and diode plus 51 ohms to ground on encoder input.
3. Differential input model is terminated with a 150 ohm resistor in series with a 4.7nF across each differential encoder pair.

7.2 Communicating with the Driver Directly

7.2.1 Getting Started

US Digital provides a DLL wrapper that encapsulates many of the commands, but a direct access method to the driver is also possible. There are benefits to this approach, as the most direct way to exchange data with the USB1 product is to send commands directly to the driver through Windows API, using calls such as CreateFile, DeviceIoControl, and CloseFile.

Under the Windows API, devices must be opened in the same manner as disk files before they can be used. To accomplish this, call CreateFile with the symbolic link name of the USB1 device to be talked to. The driver assigns a symbolic link name to each device as it is plugged in; the first device becomes “USDUSBB0”, the second device becomes “USDUSBB1”, etc. If only one device is ever going to be plugged into the bus, the user can just call CreateFile with the parameter “USDUSBB0”; however, it is good defensive programming to allow for more than one device by attempting to open each of the instances from 0 to 9 (the maximum possible), and cataloging the possible presence of each device.

A typical call to CreateFile is shown below:

```
lngFileHandle = CreateFile(strFilename, GENERIC_WRITE, FILE_SHARE_WRITE, NULL,
OPEN_EXISTING, 0, NULL);
```

When the program is done using the device it should call CloseFile() with the file handle obtained above.

If there are multiple USB1 devices on the bus the user will need some way to relate a physical device instance to a logical device identity in software. Each USB1 has a unique serial number, so that is one way to relate logical device to physical device. To make this work, the user must open each of the devices on the bus, and then query it for a serial number.

There is a second method of relating physical and logical devices. Open the device as described above, but then query the device for its module address number. The module address is simply a register that can be set, read, and saved in non-volatile memory. The advantage of using a module address is that it then becomes possible to replace a given USB1 device with another USB1 device without requiring an adjustment to the software to account for the fact that serial numbers are different. The software may not care if the modle address is correct. The demonstration program USB1_Explorer has a command to allow the user to set the module number.

The driver allows more than one application and more than one thread to open access to a USB1 device, and it arbitrates the requests so that data integrity is guaranteed.

Once the device has been opened, the program communicates with the driver via DeviceIoCtrl calls. Note that unlike the disk file, the USDUSB1 driver does not support Read or Write requests. The device firmware works on a command-response sort of model, and the DeviceIoCtrl paradigm fits this model well: when the DeviceIoCtrl is invoked to call passes in a buffer containing the command to the USB1, and the driver puts the response from the USB1 in the output buffer supplied.

7.2.2

Embedded within the actual device I/O control codes is the file system, the actual index of the command, and the method of access. However, all the software needs to know is which code talks with which subsystem. The table below shows the codes that apply to each subsystem:

Incremental and general control	0x222084
SEI host subsystem	0x222094
Interrupt polling subsystem (for recent position)	0x222098
Buffered data subsystem	0x22209C
Flush buffered data buffer	0x2220A0

Note: Sending other control codes to the driver may result in unusual behavior and is not advised.

7.2.2 Incremental & General Control Subsystem

There are four subsystems on the USB1 that can exchange data with each other, and each is accessed with a separate DeviceIoControl code. The first one is the general device control, including the incremental encoder interface. The second is the SEI bus host processor. The third is a specialized portion of the device firmware that supports USB interrupt polling; calls to this subsystem return a short summary of encoder positions, parallel input data and a timestamp. The fourth is similar to number three, in that it returns data for all four encoders, a timestamp, and the parallel input data, but it is buffered within the driver so that each call returns the next frame of data; it thus provides a buffered history function, with samples occurring at regularly programmed intervals. Each of the four subsystems may be thought of as a separate command processor, and each has its own DeviceIoControl code that can be used to send a command and receive a response back. Commands may be sent asynchronously and simultaneously; the driver serializes access to the hardware and sorts out who gets what result. In addition, there is a fifth DeviceIoControl code that resets (flushes) the history bufffer. A typical call to DeviceIoControl might look like the following:

```
lngResult = DeviceIoControl(lngFileHandle, _
          IOCTL_EZUSB_USD_COMMAND,   // Device I/O control code
          bytCommandBuffer,               // pointer to buffer
          lngCommandLength,
          bytResponseBuffer,         // pointer to buffer
          lngResponseLength,
          &lngBytesReturned,         // pointer to long to hold number of bytes
     returned
          NULL)
```

The device I/O control code specifies which subsystem is being addressed. The command buffer is the input for the subsystem, and the USB1 puts its output in ResponseBuffer. The CommandLength parameter should be used to tell the USB1 how many bytes were placed in the command buffer. The ResponseLength parameter may be set to the actual buffer length (except in the case of the SEI subsystem, see below), and not necessarily the actual length of the command and response. The buffers can be defined as 64 bytes in length and then hardcode the response length as 64L if preferred, as the driver just uses it to make sure the buffer is not read or written past the end of the buffer. The variable lngBytesReturned is where the driver will tell the software how many bytes were actually returned by the device.

The call to DeviceIoControl will return non-zero if the transmission was successful, and zero if it was not. Within the result returned by the USB1 there may be a status code as well, indicating whether the USB1 was successful in processing the command it was sent.

The incremental encoder/general device subsystem supports a variety of commands. A typical command might be the Read Position command; to use this function two bytes need to be placed in the command buffer: 0x01 0x00 (to read channel 0, for instance), the USB1 will send 14 bytes back to tell the position, the current counting mode, the timestamp

of the data, and some other parameters. The software would be responsible for passing the output from the USB1 into the various fields. The document “USB1 command protocol.doc” has the details on each command supported by the USB1, and communicates what the device returns for each command sent to the incremental subsystem.

Set Timestamp
1 byte: 0x39
The four bytes following the command code (positions 1 [MSB] through 4 [LSB] in the command buffer) are treated as a value to be assigned to the internal timestamp register.

7.2.3 General Device IO Control Commands

The following list represents the raw commands understood by the firmware of the USB1 device. To communicate with the USB1 using these commands, place the requested data bytes in a buffer and pass that buffer to the driver via the general device IO control command code. The driver, in turn, will transmit the command buffer to the device over USB, and return to the code a buffer of data from the device. Every command generates a response; most commands return 1 for success, 0 for failure.

Note also that many of the commands interact closely with the LS7266 encoder counter chip, and in some cases the LS7266R1 data sheet may need to be consulted to interpret the data being sent or received.

The timestamp is a 32 bit value that increments every 100 usec.

Read channel data
2 bytes: 0x01, channel number (0-3)
Returns 14 bytes: 4 bytes position (MS to LS byte), 4bytes maximum count (MS to LS byte), 4 bytes time stamp data (MS to LS byte), counter mode byte, flag/status byte from the 7266 encoder channel.

Reset channel position
2 bytes: 0x03, channel number (0-3)
Returns 0x01 for success, 0x00 for failure.
Resets the specified channel to 0.

Reset all four positions
1 byte: 0x04
Returns 0x01 for success, 0x00 for failure.
Resets all four incremental channel positions to 0.

Set channel position
6 bytes: 0x05, channel number (0-3), 4 bytes position (MS to LS byte)
Returns 0x01 for success, 0x00 for failure.
The channel position requested must be a positive integer number less than 16,777,216.

Set channel maximum count
6 bytes: 0x06, channel number (0-3), 4 bytes maximum count (MS to LS byte)
Returns 0x01 for success, 0x00 for failure.

The maximum count is typically used to specify where the 7266 counter used by the USB1 should roll over. The default maximum count is 3600, which in practice means that the counter will usually be set up to roll over from 3599 to 0000. This command also has the side effect of resetting a channel’s counter to 0. The channel maximum count requested must be a positive integer number less than 16,777,216

7.2.3

Set channel counter mode

Four bytes: 0x07, channel number (0-3), counter mode byte, prescale value
Returns 0x01 for success, 0x00 for failure.
The 7266 counter mode controls how the counter behaves on overflow and underflow, as well whether it increments on each edge or each cycle; it is set to 0x2E by default, resulting in modulo-N, X1 quadrature. The prescale value is loaded into the counter's digital filter and determines the highest allowable quadrature input frequency that will be recognized; a typical value is 0x0E. The formula for maximum allowable quadrature input frequency as a function of the prescale value is:
Max input = (12 MHz)/((prescale+1) * 8)

The table below tabulates some common values of prescale and the resulting maximum quadrature input frequencies for the USB1:

0	1.5 Mhz
1	750 kHz
2	500 kHz
3	375 kHz
4	300 kHz
6	214 kHz
9	150 kHz
14 (0x0E)	100 kHz

See the LS7266R1 documentation for a more detailed explanation of these parameters.

Set SEI baud rate

2 bytes: 0x10, baud rate byte
The USB1 SEI port supports 6 different baud rates specified by the value of the baud rate byte, as shown in the table below:

Baud Rate Byte	Baud Rate	Timing Error
0x01	57600	+1.16%
0x10	38400	-2.34%*
0x11	19200	+0.16%
0x12	9600	+0.16%
0x13	4800	+0.16%
0x14	2400	+0.16%

* May not work in all systems due to inaccuracy.
The USB1 always powers up at 9600 baud; any changes to the baud rate persist until a master reset or the unit is powered down. Note that the devices on the SEI bus must first be changed to a new baud rate, and the USB1 programmed for the new baud rate after the checksum has been received from the SEI devices. These baud rate codes match the codes supported by other US Digital SEI devices, though they represent a subset of the usual baud rates supported. Sending an invalid baud rate byte will result in no change to the baud rate.
Returns 0x01 for success, 0x00 for failure.

Send SEI break

1 byte: 0x11
The unit will immediately send a 1 second break condition to the SEI bus, which will reset all devices on the bus.
Returns 0x01 for success, 0x00 for failure.

Read SEI status
1 byte: 0x12
Returns 1 byte of SEI status: if bit 0 is set, there was a framing error; if bit 1 is set, was an over-run error in which the USB1 could not keep up with the incoming character stream, and some data was lost. Bit 2 (mask value 0x04) is set if the busy lines are currently asserted on the SEI port.
Reading the SEI status clears any errors accumulated and reported.

Read factory info
1 byte: 0x20
Returns 14 bytes: 2 bytes model number, 2 bytes version, 2 bytes configuration, 4 bytes serial number, month, day, 2 bytes year (all MS to LS byte order)

Read index enable register
1 byte: 0x21
Returns 1 byte from the index enable register.
The lower nibble of the byte returned represents whether each of the 4 channels allows an external index strobe to reset or load the counter.

Set index enable register
2 bytes: 0x22, index enable byte
Returns 0x01 for success, 0x00 for failure.
The lower nibble of the byte sent controls whether each of the 4 channels allows an external index strobe to reset or load the counter. A bit set means external control is allowed.

Save power-up parameters to non-volatile storage
1 byte: 0x23
Upon reception of this command, the USB1 will save counter modes, maximum resolutions, module address, and index enable register to non-volatile EEPROM storage. The saved data will be used to restore the state of the device at the next power-up. Note that each of the 4 internal counters for incremental encoders starts at 0 on power-up; encoder positions are not saved. This command takes serveral miliseconds to complete, and should not be a preformed when data is being logged via the USB1 history mechanism.
Returns 0x01 for success, 0x00 for failure.

Read input port
1 byte: 0x24
Returns 1 byte of data from parallel input port.

Set output port
2 bytes: 0x25, 1 byte of data destined for parallel output port.
Returns 0x01 for success, 0x00 for failure.

Read module address
1 byte: 0x26
Returns 1 byte module address.

Set module address
2 bytes: 0x27, 1 byte module address
Returns 0x01 for success, 0x00 for failure.
The module address set in this command is volatile, and does not persist when the device is reset. To make the module address permanent, perform a “Save power-up parameters to non-volatile storage” command.

7.2.3

Read output port setting
1 byte: 0x28
Returns the most recent data sent to the parallel output port.

Read user EEPROM area
3 bytes: 0x29, 1 byte starting address (0-63), 1 byte length (1-64)
A 64 byte area of the EEPROM has been set aside for use by user applications; this command provides the read access to that area. For example, to read the entire user area of 64 bytes the user program would issue a call with the command string 0x29 0x00 0x40; to read just the last byte (where a checksum might be stored, for example), the user program would issue a call with the command string 0x29 0x3F 0x01. The user area does not wrap around, attempts to read or write past location 63 will return an error.

Write user EEPROM area
3 byte header, then data: 0x2A, 1 byte starting address (0-63), 1 byte length (1-61), then up to 61 bytes of data.
This command always returns status of 0x01 for a correctly formatted command, whether or not the processor has reason to believe that the command actually succeeded. Use the “read last status” command to query the true status of the result.

This command writes byte data into the 64 byte user EEPROM (non-volatile) memory. Note that a maximum of 61 bytes may be written at a time; in order to completely fill the user EEPROM area the program will have to issue 2 commands. There are no restrictions on where in the user area writing has to begin, or on the length of data written, provided that the user does not attempt to write past location 63 or write more than 61 bytes at a time. The format of the data written is also entirely up to the program.

Read status of last command
1 byte: 0x2B
This command returns the status (0x00 or 0x01) of the last command processed. This is useful for a couple of situations. Some commands simply return a single byte, but it is not possible to know whether that byte was status or the state of an input. Other commands, like those that write to the EEPROM, always return success status for a correctly formatted command so as not to stall the driver, when in fact there may have been an error. This command will fetch the true status of the last command.

Set output bits
2 bytes: 0x2C, then a mask of bits to turn on.
Returns 0x01 for success, 0x00 for failure
This command provides a way for a user application to turn on a specific bit or bits of the parallel output port without having to read the output port to find out the condition of the unaffected bits. Bits set in the mask cause the corresponding bits of the output port to be activated, sinking current. Bits not set in the mask retain their current value on the output port.

Clear output bits
2 bytes: 0x2D, then a mask of bits to turn off.
Returns 0x01 for success, 0x00 for failure
This command provides a way for a user application to turn off a specific bit or bits of the parallel output port without having to read the output port to find out the condition of the unaffected bits. Bits set in the mask cause the corresponding bits of the parallel output port to be de-activated, or stop sinking current. Bits not set in the mask retain their current value on the output port.

Reset history buffer
One byte: 0x30
Returns 0x01 for success, 0x00 for failure
Clears the history buffer of all accumulated samples, so that history logging will start anew. Note that this affects only the buffer within the USB1 device itself; the user must clear the driver buffer separately via an DevIoCtrl call.

Get history buffer status
One byte: 0x31
Returns 0x01 if no buffer overrun, 0x00 if buffer has been overrun
This command is provided as a means of checking the integrity of the history logging process – if the firmware has not been able to send all the samples over the USB interface without losing samples, this command will return a 0 for failure. The status is reset to no overrun after reading this command.

Set history logging sample rate
Five bytes: 0x32, four bytes sample rate (MS to LS byte)
Returns 0x01 for success, 0x00 for failure
The 32 bit number supplied is used to decimate the 1 ms sample stream before it is put in the USB pipeline. Thus a 0x00000005 parameter will result in history samples that are 5 ms apart. A 0 parameter is not allowed, and will result in the USB1 returning a 0x00 response. The change will take effect immediately after the command is received. Note that read position requests are not affected, and immediate position information is still sent continuously to the driver in every USB frame while a device file request is open.

Get history logging sample rate
One byte: 0x33
Returns a 4 byte long that represents the decimation rate for samples in units of milliseconds.

Set event triggering and qualification
23 bytes: 0x34, 8 bytes channel trigger, 1 byte trigger AND/OR selector, 8 bytes channel qualifier, 1 byte qualifier AND/OR selector, 4 byte unsigned long representing number of samples to collect.
Returns 0x01 for success, 0x00 for failure.

The USB1 supports a fairly sophisticated (albeit low speed) triggering and qualification system for the history logging function. The user may define a trigger that the USB1 will search for before entering the data storage phase; once in the data storage phase, the user also defines a storage qualifier that must be met each and every time that data is considered for storage. Any or all of the bits may contribute to the logical expression that determines when the unit triggers or stores; each bit's contribution may be defined to be one of several types of conditions, and the whole group of defined bit contributions may be ANDed or ORed together to form the final result. The conditions that the USB1 will recognize are listed in the table below:

Name	Description	Code
Never (Ignore)	The affected bit trigger or qualifier is never true (does not participate).	0
Rising	The affected bit trigger or qualifier is true when a rising edge is seen.	1
Falling	The affected bit trigger or qualifier is true when a falling edge is seen.	2
Change	The affected bit trigger or qualifier is true with either a rising or a falling edge is seen.	3
High	The affected bit trigger or qualifier is true when a logic high is seen.	4
Low	The affected bit trigger or qualifier is true when a logic low is seen.	5
Always	When chosen, the affected bit trigger or qualifier is always true.	6

7.2.3

The bytes that define the trigger condition consist of a eight separately coded byte condition codes, one for each channel beginning with channel 0, followed by a byte that determines whether the preceding conditions are to be ANDed together (non-zero) or ORed together (zero). Thus a string of bytes like “0 0 0 1 0 0 0 5 1” means that bits 0, 1, 2, 4, 5, 6 do not participate in the trigger, and the trigger itself consists of a rising edge on bit 3 when bit 7 is low (the non zero final byte makes the condition an AND of all the defined bit contributions).

The storage qualifier description has the same format: eight coded byte contributions, with a ninth byte that determines whether the overall structure of the qualifier is an AND or an OR.

The final four bytes determine how many samples are collected before the system stops logging and waits for another start acquisition command. If sending a parameter of 0 the USB1 assumes that it should collect data continuously until a Stop Acquisition command is issued.

The event triggering and qualification system uses the data collected from the parallel port at the sampling rate of 1 kHz; any event that is to be recognized must therefore exist for longer than one millisecond.

The trigger and storage qualifier are evaluated on every 1 millisecond frame, regardless of the sample period chosen. The sample period specified will interact with the storage qualifier; however, the sample period specified with command 0x32 acts as a decimation rate on the qualified storage events. Thus if a sample period of five milliseconds is specified and the storage qualifier is set to store when b0 is low, the system will store every fifth sample in which it finds b0 low, checking b0 every millisecond. If b0 is low continuously for long periods, data will be stored effectively at a 5 millisecond sample period, but if b0 goes up and down a lot there may be slightly different results.

Note that the unit starts up in a triggered state with every bit always sets to storage qualifier so as to enable simple history logging without defining a trigger or qualifier. It is not enough to simply send a new trigger/qualifier definition. A Stop Acquisition command must be issued to be able to acquire a triggered acquisition, since the device already thinks it has triggered.

Get event triggering and qualification
One byte: 0x35
Returns 22 bytes: eight bytes trigger definition, one byte AND/OR trigger structure, eight bytes qualifier definition, one byte AND/OR qualifier structure, 1 4-byte long showing number of samples to be collected. See previous command for definition of trigger and qualifier information.

Start acquisition
One byte: 0x36
Always returns 0x01 for success.
Resets the trigger system to the untriggered state, and begins searching for a specified trigger. Once the trigger has been recognized, it will store samples that meet the storage qualification criteria specified until the specified number of samples have been collected.

Stop acquisition
One byte: 0x37
Always returns 0x01 for success
Stops an acquisition in progress.

Get acquisition status
One byte: 0x38
Returns nine bytes: one unsigned long representing the total number of samples to be collected, one unsigned long representing the number of samples remaining to be collected, and a status byte. The status byte definitions:

bit 0	When set, an acquisition is in progress and has not completed.
bit 1	When set, the acquisition has been triggered.
bit 2	When set, the acquisition in progress is a continuous acquisition.
bit 3	When set, the acquisition subsystem failed to keep up with the sample clock, and a data point may have been missed.

Set Timestamp
One byte: 0x39
The four bytes following the command code (positions 1 [MSB] through 4 [LSB] in the command buffer) are treated as a value to be assigned to the internal timestamp register.

Get Encoder Channel Trigger/Qual Enable Byte
One byte: 0x3A
Returns one byte with bit 0 set if encoder channel 0 is enabled to participate in triggering or storage qualification, bit 1 set for channel 1, etc.

Set Encoder Channel Trigger/Qual Enable Byte
Two byte: 0x3B, enable byte
The enable byte supplied is used to define which encoder channels are enabled to participate in triggering and storage qualification. Setting bit 0 enables channel 0, setting bit 1 enables channel 1, etc. Note that encoder channels are allowed to trigger or store only when all the digital input bits are set to ignore in either the Trigger or Qualification.

7.2.4 SEI Host Subsystem

The SEI host subsystem functions strictly as a conduit to the SEI port on the USB1. No interpretation of the data occurs en route; gibberish can be sent out of the SEI port, and the USB1 will not know or care. The lngCommandLength parameter to DeviceIoCtrl is important. This is how the USB1 determines how many bytes to send out to the SEI port. The format of the bytes sent out and the response received back depends on what the software is talking to on the SEI bus. The bytes sent via the USB1 should be the same sequence that would be sent out through a COM port to talk to an equivalent SEI bus setup. Thus, if an A2 absolute encoder has been connected to the SEI bus of the USB1, and it has an SEI address stored in it of 3, send it a command 0xF3 0x03 via the bytCommandBuffer to cause the A2 to return its serial number. In this case the software will get 5 bytes back in the bytResponseBuffer; four serial number bytes, plus the checksum byte from the A2.

The SEI host subsystem has a timeout timer, if there is no device present, the DeviceIoCtrl call will return with an error. The driver always expects to get something back from the SEI bus; this can be tricky when talking to an SEI device that indicates failure by withholding an acknowledgement byte (the A2 Get Address function is one such instance).

The USB1 does not interpret the data, all the data is sent as fast as possible with no interruption. Recall that the SEI bus is half-duplex; only one party can be transmitting over the bus in one direction at a time. Therefore, be sure to format only a single SEI command into any given packet sent via this method, otherwise, the first command will get sent, the SEI remote unit will try to respond, but the USB1 will continue sending the remainder of the packet containing the second command. Data will almost certainly be lost as both the USB1 and the responding device try to talk on the bus at the same time.

7.2.5

The USB1 firmware collects the data received from the SEI bus; software keeps track of what has been sent, and throws away received characters that are simply echoes of the characters sent out by the USB1. The firmware groups received characters together in packets for maximum efficiency; it uses a heuristic rule to decide when to close out a packet and transmit it up to the host over USB. In general terms, it closes out a pending packet when there have been no further received characters for 10 milliseconds.

7.2.5 Buffered Data (Interrupt Polling) Subsystem

The USB1 driver returns position data through one of two methods: interrupt polling or buffered data. Under interrupt polling the driver requests data from the device periodically, bypassing the command-response mechanism used for most parameters. We have set up the USB1 to supply fresh position and timestamp data to the driver every millisecond. The driver stores this data away in a safe place, and updates it as fresh data is delivered. User programs may use the interrupt polling device I/O control code to request this data; by having fresh data pre-delivered on every frame the latency for user programs is minimized. Calls to the USB1 DLL that invoke USB1GetAllPosition() or USB1GetIncPosition() also use this interrupt polling feature to minimize data latency.

The other method that the USB1 driver uses to get data is the buffered method. This method is used to retrieve position data sequentially without gaps. It does this at the expense of latency, but the resulting data is useful for reconstructing a record of position versus time without gaps in the data. The record of data returned via this method may not actually be the most recent reading of position, but it is guaranteed to be the next record in the sequence of readings. This is the method used by the data logging calls to the DLL; USB1GetBufferedPosition() and USB1GetAllBuffered() use this method.

7.2.5.1 Interrupt Polling Details

The command length for the interrupt polling device I/O control command is a don't-care value; you may supply zero. The driver always returns twenty-one bytes of information; the first sixteen bytes comprise four long 32 bit words that hold the position data from each of the four incremental encoders, the next four bytes are for the timestamp, and the last byte is the data read from the parallel I/O port on J2. Note that in this area, as well as other cases where the encoder returns multi-byte data, the format of the data is big-endian, with most significant byte in the lowest address. When communicating directly with the driver your code is responsible for swapping the bytes end for end to match the little-endian format expected by Intel/Windows processors.

7.2.5.2 Buffered Data Details

Calls made with the buffered data IOCTL return data in the same format as the interrupt polling IOCTL, but instead of getting the most recent data the driver returns historical data from a circular buffer. The circular buffer has a write pointer and a read pointer; as new data arrives from the device the data is written into the local buffer and the write pointer is advanced. As an application reads data from the local driver buffer the read pointer is advanced. The "bytes returned" field of your DeviceIoControl call will return with one of two values: it will return 21 if there was data placed in the output buffer, or it will return 0 if no data is currently ready in the driver buffer (the application has caught up with the write process). If an application fails to keep up with the write process, and the buffer becomes completely filled, the read pointer is advanced automatically so that subsequent reads begin one buffer length back from the current write pointer location; in this situation data will be lost. The size of the buffer is 4096 slots, which gives more than four seconds of buffering. If a DeviceIoControl call to the buffered data subsystem returns an error it is probably because the output buffer you supplied wasn't big enough.
The default sample rate for the buffered history function is 1 kHz, but there is a command to specify the sample interval in units of milliseconds.

We recommend that you issue a "flush buffer" device IO control code prior to starting the process that retrieves data from the buffered data subsystem. Otherwise, the first data that you fetch will be data that was collected four seconds ago, from the period when the buffer was full. Starting with an empty buffer also helps ensure that if the application fails

to keep up with the write process momentarily, no data will be lost. The flush buffer device IO control requires no data in the command buffer, and returns no data in the response buffer.

7.3 Power-on Self Test

Beginning with driver/firmware version 1.07, the USB1 incorporates a power-on self test sequence. The unit will flash all its LED's once, and then begin testing various subsystems of the device. As it does this, it also lights a characteristic LED associated with the subsystem under test. The tests and corresponding LED's are listed below:

Test	LED
Tests bit independence of data bus	Channel 0 activity LED
Tests bit independence of adress bus	Channel 1 activity LED
Tests every byte of memory exhaustively	Channel 2 activity LED
Tests the EEPROM for proper checksum	Channel 3 activity LED
Tests the encoder counter chips	USB activity LED

The net effect is that the user will see a brief flash of all the LED's that signals the start of testing, and then the channel LED's will each light in sequence, followed briefly by the USB LED. After a period of about two seconds the channel LED's will flash simultaneously and then extinguish. The USB1 is operating normally at this point.

If the USB1 detects a problem it will jump out of the test sequence and flash a diagnostic code three times. In between code flashes it flashes all the LED's to signal failure. Following this it will attempt to run firmware as usual, but it may or may not be able to function properly, depending on the nature of the error. If this condition is encountered please contact US Digital Technical Support for assistance.

7.4 Frequently Asked Questions

1) If the device reports position to the host every millisecond, how can it keep up with quadrature inputs of several hundred kilohertz?

Quadrature inputs are tracked in hardware, which can keep up with very fast signals. The one millisecond (or one kilohertz) update rate is the rate at which those counters are read and reported back to the host for use by applications.

2) What steps have been taken to make the position measurements low-latency? What is the latency from the time an input changes to the time a valid position result is received by the application?

The driver is set up to stream data from the device into the host whenever a device is in use (i.e. a file handle is open to the device). This ensures that up-to-date information is readily available in the driver data structure before the application even asks for it.

Measurements made on a PII-400 MHz Windows 98 system with 6 inactive background programs suggest that the latency averages about 2.2 milliseconds from the time an input changes to the time that the application notices the change. Best case latency was just over one millisecond, with worst case at 3.5 milliseconds. 80% of the readings were below 3 milliseconds.

3) How many USB1's can I connect to my system?

Our observation is that a single USB controller can support up to four USB1's; beyond that point, Windows recognizes the device, but fails to award it any pipes, indicating that there is no more reserved bandwidth available. Adding USB hubs will not increase your bandwidth, and is not a solution. However, you may be able to access additional USB1 devices by installing an additional PCI-to-USB interface card, and plugging USB1's into that. What matters is how many USB controller IC's there are in the system. There is some system-dependent point where the host computer simply can't service the real-time needs of all the the connected USB1's; this number varies, from four to six or so.